Characterizing the Performance of Tenant Data Management in Multi-Tenant Cloud Authorization Systems

Pieter-Jan Maenhaut*†, Hendrik Moens†, Maarten Decat‡, Jasper Bogaerts‡,
Bert Lagaisse‡, Wouter Joosen‡, Veerle Ongenae* and Filip De Turck†
*Ghent University, Faculty of Engineering and Architecture, Dept. of Industrial Technology and Construction
Valentin Vaerwyckweg 1, 9000 Ghent, Belgium
†iMinds – INTEC, Ghent University, Dept. of Information Technology
Gaston Crommenlaan 8 bus 201, 9050 Ghent, Belgium
‡ iMinds – DistriNet, KU Leuven, Dept. Computer Science
Celestijnenlaan 200A, 3001 Heverlee, Belgium
Email: pieterjan.maenhaut@intec.ugent.be

***Abstract*—Multi-tenancy leads to improved efficiency, improved scalability, and lower costs. With the recent evolution of Cloud Computing and Software-as-a-Service (SaaS) in particular, a flexible and scalable multi-tenant architecture is becoming highly important. In multi-tenant applications, each tenant has its own users and administrators and tenants even tend to be divided into multiple subtenants. As the number of tenants grows, the number of users and amount of data grows, thus a scalable architecture for the access control system is needed. The question arises how to distribute the users and data over multiple database instances.**

In this paper we present a hierarchical data management approach, taking performance metrics into account, for structuring the storage of tenant data in large multi-tenant environments. We introduce a logical representation of the tenants, the tenant tree, and make a mapping to the physical storage by introducing three models for load-balancing. Next, we focus on how to efficiently locate the required data and introduce multiple search approaches. We characterize the impact on the performance both theoretically and experimentally. Experiments confirm that the theoretical analysis is in line with the experimental results. When the amount of data increases significantly, dividing the data over multiple datastores in an efficient way will eliminate the overhead and lead to a performance gain, especially if most of the data is located at the leaf nodes of the tenant tree.

I. Introduction

Multi-tenancy [1] enables the serving of multiple clients or tenants by a single application instance, with isolation of each tenant's data. The major benefits include increased utilisation of available hardware resources and improved ease of maintenance and deployment. These benefits can result in lower overall application costs. In a multi-tenant architecture, a software application is designed to virtually partition its data and configuration, as illustrated in Figure 1, and each tenant works in a virtual application instance. Within the application, every tenant will typically have its own users and administrators. Some tenants may be divided into multiple subtenants, each one again having its own users. A reseller for example is a special tenant, serving multiple customers, its subtenants. The

978-1-4799-0913-1/14/$31.00

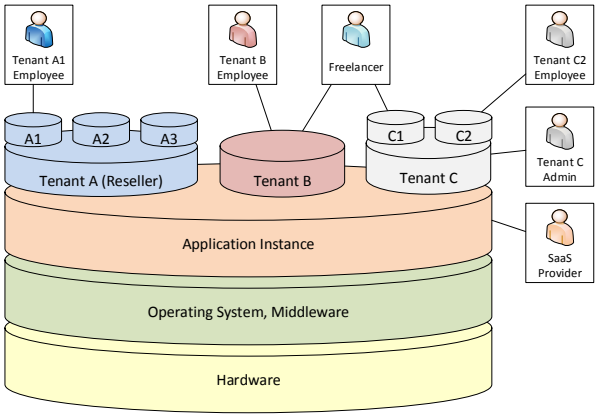


Fig. 1: In a multi-tenant application, most of the software stacks up until the application itself, which is shared by the different tenants.

PUMA project [2] aims to develop a scalable security solution for the management and enforcement of user permissions for Software as a Service (SaaS) applications in a shared (multi-tenant) infrastructure. This solution offers support for essential security requirements, such as confidentiality, integrity and availability.

With the recent evolution of cloud computing [3], a technology that enables elastic, on-demand resource provisioning, and SaaS in particular, a multi-tenant architecture has gained popularity. With cloud computing, an optimal usage of available resources is recommended to reduce operating costs, as the infrastructure provider usually charges for the number of instances used. As the number of tenants grows, a scalable architecture for authentication and authorization is needed. While most users belong to a single tenant (or subtenant), some users might belong to multiple tenants, which introduces extra challenges for a multi-tenant access control system. Examples

include a custom tenant administrator, who is responsible for multiple (but not all) subtenants, and a freelancer who works for different tenants.

Performance is a key challenge in multi-tenant environments, because multiple tenants share the same resources and hardware utilisation is higher on average, and one tenant might clog up resources, compromising the performance of all other tenants. Scalability is another big challenge, especially when the number of tenants increases [4].

In this paper we focus on the scalability and load-balancing of the storage component of multi-tenant applications, in special of the access control system. We present a hierarchical data management approach, taking performance metrics into account, for structuring the different tenants and subtenants. Different physical implementations are possible, and we will shortly describe the advantages and disadvantages of each alternative. We characterize the impact on the performance both theoretically and experimentally.

We will address these three research questions: 1) How to store, load balance and find tenant data in large multi-tenant environments with minimal overhead? 2) How does the proposed model impact the performance of the application? 3) How do tenants impact each other's performance?

In the next section we will discuss related work. Afterwards, in Section III, we will present a hierarchical model for managing and storing data, users and roles. We will discuss how specific data can be searched, and provide a theoretical analysis of the impact on the performance in Section IV. In Section V, we will verify our theoretical analysis by different experiments. In Section VI, we finish with our conclusions and future work.

II. Related Work

In previous work [5], we described the steps required to migrate an existing application to a public cloud environment, and proposed a solution to add multi-tenancy to the application. We focused on the use case of a medical communications application. In this paper, we elaborate the concept of management and storage of users and roles in a multi-tenant environment, and focus on the performance and load distribution of the access control system.

Related to the PUMA project, the work of Decat et al. [6] [7] is complementary to this paper. They focus on scalable and confidentiality-aware access control management for SaaS applications from the point of view of the tenant. To achieve this, they describe and evaluate the concept of *federated authorization* in which authorization is externalized from the SaaS application and centralized at the tenant [6]. To improve performance, they also describe a policy decomposition algorithm for more fine-grained policy deployment [7]. In this paper, we focus on the performance of the storage part of the application.

Calero et al [8] describe an authorization model suitable for cloud computing in which hierarchical role-based access control, path-based object hierarchies and federation are supported. The model described can be used to implement authentication and roles in a multi-tenancy environment, but no details are added about where to store the users and roles, especially in large, scalable environments. By contrast, we focus on how to divide the users over different datasets, and how this will influence the performance of the authentication mechanism.

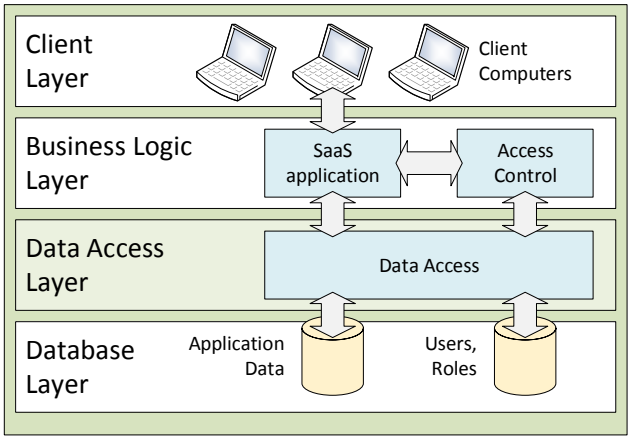


Fig. 2: Layered architecture with decoupled access control. The data access component is responsible for selecting the correct datastore.

In [9], the design of the Force.com multi-tenant internet application development platform is described. The storage uses a set of metadata, data and pivot tables to store all tenant data generically. Typically, a single database is used for every tenant. The paper presents a very generic way for storing custom objects and custom data, which could be used to store the tenant's data, but doesn't really focus on the scalability. By contrast, we present a scalable model where multiple tenants can share a single database, and characterize the performance of the model.

In [10] a solution for access control in cloud environments is presented. Access policies based on data attributes are used to enforce authorization. Such could be used to encrypt the tenants data, combined with the hierarchical model presented in this paper.

Walraven et al [11] described an architecture of a multi-tenancy enablement layer, which can be used for data isolation, feature management and tenant-specific customizations. This layer could be extended with the hierarchical model presented in this paper to increase scalability and performance, for building a middleware for highly scalable multi-tenant applications.

III. Architecture Outline

Web applications are usually designed using a multitier architecture, where the application is separated into multiple layers, as illustrated in Figure 2. Within the business logic layer, security is provided by the access control component. This component should be decoupled from the offered services as much as possible. The database layer holds the application data, the different users and (if applicable) roles. As an alternative for roles, Attribute-Based Access Control (ABAC) [12] could also be used. The reasoning behind ABAC is that every

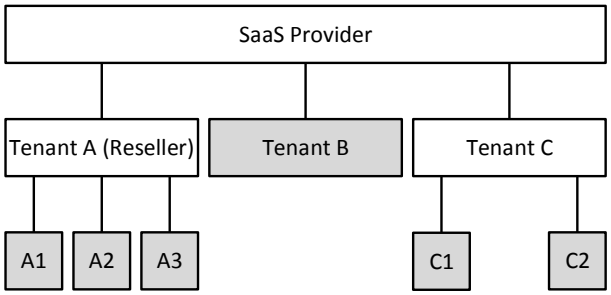


Fig. 3: The tenant tree, a logical representation of different tenants and subtenants in a multi-tenant application. Grey nodes are leaf nodes of the tree.

user, resource and action can have certain attributes related to them on which policies can define restrictions.

In this section, we focus on the scalability of the database layer in multi-tenant applications. We will introduce a logical hierarchical representation of the different tenants and subtenants, and make a mapping to the physical storage. The data access layer is responsible for load balancing between the different datastores and holds the decision support for splitting or merging datastores when applicable.

A. Logical Representation

Tenants and subtenants can be structured hierarchically. In the rest of this paper, we will refer to this representation as the tenant tree. At the top level is the SaaS provider, which can be seen as the root node of the tenant tree. The different tenants using the application are located on the next level, and can be seen as child nodes of SaaS provider. Therefore, all tenants share the same parent. Some tenants can even be divided into multiple subtenants, for example in the scenario of a reseller. In this case, the subtenants are child nodes of their respective parent tenant, making the tenants inner nodes (nodes with child nodes) of the tree and the subtenants leaf nodes (nodes without children).

Figure 3 shows an example tenant tree where a multi-tenant software application, deployed on the public cloud by the SaaS provider, is used by three different tenants. All tenants have the same parent, the SaaS provider. Tenant A, a reseller, has three child nodes, its clients, while tenant C has two child nodes. Subtenant A1 and tenant B are examples of leaf nodes (colored in grey), whereas tenant A is an example of an inner node. Inner nodes also have some tenant data.

B. Physical Storage

By introducing the logical hierarchical representation, the question arises how and where to store the data for the different tenants. Data could be split in the same way as the logical representation, by creating a datastore for each (sub)tenant or merge multiple smaller databases. For small applications with a limited number of tenants, a single datastore can be used. We distinguish 3 different models for load balancing the data:

1) The **monolithic model**, where all data is stored in a single datastore.
2) The **fully distributed model** where every tenant and subtenant has its own datastore.
3) The **hybrid model** which is a mix of both previous models.

TABLE I: Comparison between the monolithic and fully distributed model.

	Monolithic model		Fully distributed model	
Cost	+	single instance, cheaper	-	multiple instances needed, higher cost
Implementation	+	easier to implement, search less complex	-	application needs to support multiple datastores, search more complex
Security	-	only on application level	+	data isolation, security both by application and database
Performance	-	shared resources, single tenant can clog-up application	+	dedicated instance for every tenant
Scalability	-	only usable for limited number of tenants	+	highly scalable

Table I shows a comparison between the monolithic and fully distributed model. For a new SaaS application with a limited number of tenants, the provider could start using the monolithic model for storing all data, and move to the hybrid model or the fully distributed model as the amount of tenants and data grows. The monolithic model will be easier to implement with lower costs, as only a single database instance is needed. The fully distributed model on the other hand is more flexible and scalable, with guaranteed data isolation, but at a higher price. A good architecture should support both models, making it possible to select the optimal strategy for every tenant. The application provider could select the optimal strategy himself, or leave the choise to the tenant. In the latter case, the provider could offer different the application in different versions with a different service-level agreement (SLA), for example basic hosting using the monolithic model, silver hosting using the hybrid model with a single datastore for a tenant and its subtenants, and gold hosting using the fully distributed model.

When storing data from multiple tenants into a single datastore (monolithic or hybrid model), each data row should be accompanied by a unique identifier for the tenant, the *tenantId*. In addition, add a table containing information about all different tenants and their corresponding *tenantId*. When speaking about tenant data, we can make a distinction between:

1) The tenant **users**, where all users belonging to a single tenant are stored.
2) The tenant **roles**, where the different roles for the tenant users are stored.
3) The tenant's **application data**, specific for the SaaS application.

Different combinations of the storage models for each data type are possible. Figure 4 shows an example mapping between

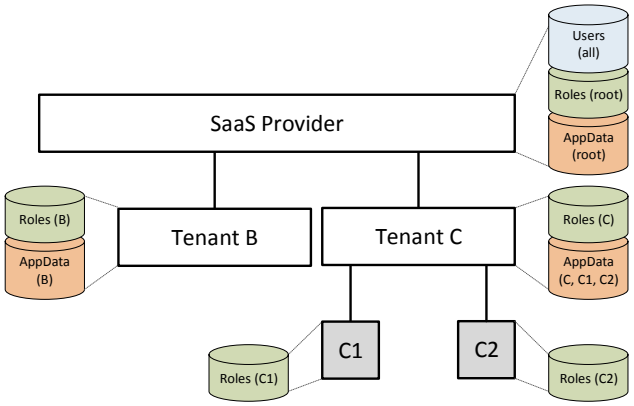


Fig. 4: Mapping between a part of the tenant tree and the physical storage of the data for the different tenants and subtenants.

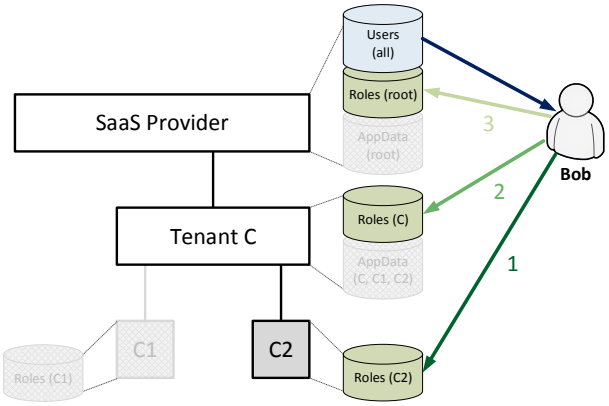


Fig. 5: Example scenario where user "Bob" wants to access the application data of subtenant C2. The required role can be stored at different locations in the tenant tree.

the the tenant tree and the physical storage of the different types of data. In this example, all tenant users are stored using the monolithic model, the application data of tenant C using a hybrid model, and the roles using the fully distributed model. The application data of tenant C and subtenants C1 and C2 is stored in a single colocated datastore.

Apart from selecting the strategy for splitting the data, which data do we store in which datastore? The main goal is to store the data at the lowest possible node of the tree, starting at the root node. For example, when the roles are stored using the fully distributed model, as in Figure 4, the subtenant-specific roles are stored at the subtenant datastores. A user who only belongs to subtenant B1 will have a corresponding role in the roles datastore of subtenant B1. A tenant administrator, who manages all subtenants of tenant B, will typically have a tenant administrator role stored in the roles datastore of tenant B. A tenant administrator, who manages some (but not all) subtenants of tenant B will have a custom administrator role, also stored in the roles datastore of tenant B. Alternatively, we could give the last user separate roles in the datastores of the different subtenants, but this introduces a small overhead as multiple roles are needed for a single user.

IV. Distributed Search

By introducing a hierarchical model for users, roles and tenant data, some data may be distributed over multiple database instances. This will have an impact on the performance and scalability of the application, as the number of users and amount of tenant data can now be much higher, but the system might have to search in multiple databases.

The question arises how to efficiently retrieve the needed data. In this section, we will propose multiple search methods, followed by a theoretical analysis of the impact on the performance.

Figure 5 shows an illustrative scenario of a fully distributed model, with a tenant C and subtenant C2. An authenticated user "Bob" wants to access the data of subtenant C2. The authorization system needs to know wether "Bob" has access to the data of subtenant C2, or more concrete, if this user has an applicable role for this subtenant. Because the model is fully distributed, meaning that every tenant and subtenant has its own roles datastore, the corresponding role for user "Bob" can be stored in three locations, as illustrated in Figure 5:

1) The roles datastore of subtenant C2.
2) The roles datastore of tenant C.
3) The roles datastore of the SaaS Provider (the root).

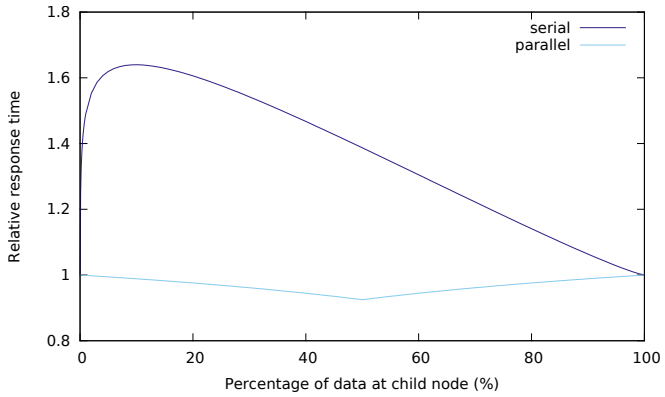
Although this example scenario only handles the search for a role in the roles datastore, a similar scenario can be described for searching a user or finding some tenant data.

A. Search Methods

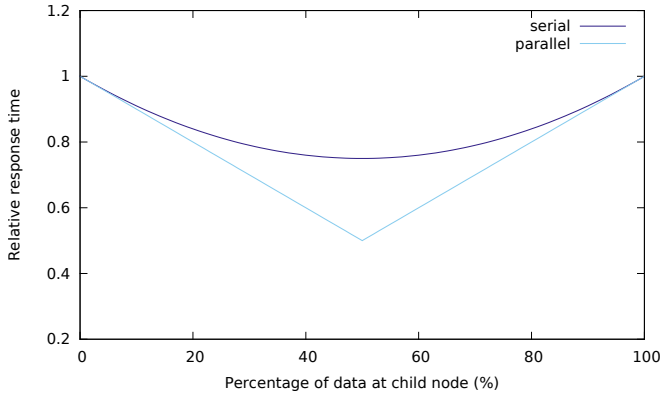
When searching for the data, in our example scenario a corresponding role for user "Bob", we need to search at different locations. In general, the data can be stored at any location along the path from the (leaf) node to the root. We can do a *serial search*, starting at a single datastore, e.g. the leaf datastore, and moving to the next datastore along the path in case the data is not yet found, or in case the datastores are stored on different (virtual) machines, we could also perform a *parallel search*), searching in all datastores at the same time and merging the results.

In case of the *serial search*, we can start at the leaf node, and move up the tree towards the root node, this is the *bottom-up* approach, or we could start at the root and continue down the tree towards the leaf node (*top-down* approach). However, we strongly prefer the bottom-up approach. The bottom-up approach is easier to implement, as in the top-down approach, the path from the root to the leaf node needs to be calculated in advance. Also, when using the top-down approach, all traffic needs to pass the root node, turning this datastore into a possible bottleneck.

In most cases, the search can stop when an applicable role is found. In some scenarios however, roles higher in the



(a) Distributed Indexed Search



(b) Distributed Non-Indexed Search

Fig. 6: Theoretical analysis of the time needed for finding the tenant data in a 2-level hierarchical structure tenant-subtenant. The vertical axis denotes the relative response time, where 1 equals the time needed to find the data in a single datastore.

hierarchy could overrule lower roles. This will have a bad influence on the performance as the authorization system needs to search all datastores to get all roles. We strongly recommend to avoid such scenarios, as it will not only have a bad influence on the performance, but the authorization system will also become far more complex.

B. Theoretical Analysis

As the number of possibilities for dividing the data over the datastores is endless, we will breakdown the problem into two cases, which can be combined for the implemented model. The symbols used in this section are summarized in Table II. We will only focus on the distribution of users among different datastores, but the same approach can be followed for distribution of roles and/or tenant data (if applicable). When using the serial search, we will use the bottom-up search, for the above-mentioned reasons.

As described in the previous section, tenants can be logically organised in a hierarchical way, and a mapping can be made to the physical storage locations.

1) Time required to find a user in a datastore: We start our analysis with the time needed to find a single user in a large dataset. The value is dependent on both the number of users in the dataset, and the indexing method used.

TABLE II: Overview of used symbols.

Symbol	Description
t_{ds}	time to find a record in a single datastore, based on an indexed parameter
t^*_{ds}	time to find a record in a single datastore,based on a non-indexed parameter
n_{ds}	number of records in the datastore
n_{tot}	total number of records
p_{ds}	probability that user record is stored in the specific datastore

When searching for an entry in a single table, based on an indexed parameter, the average time needed has a factor $O(\log(n))$, when searching on a non-indexed parameter, this factor is $O(n)$, where n equals the number of rows in the table.

Therefore, in theory, the time needed to find a user in a single datastore will be equal to

$$t_{ds} = C \times \log(n_{ds}) \tag{1}$$

$$t^*_{ds} = C \times n_{ds} \tag{2}$$

where t_{ds} and t^*_{ds} denote the time needed to find a user (record) in a dataset, based on an indexed and non-indexed parameter, n_{ds} denotes the number of users in the datastore, and C is an unknown constant factor.

2) Probability: When we have multiple datastores, the probability that a random user u is stored in a specific datastore is equal to

$$p_{ds} = n_{ds}/n_{tot} \tag{3}$$

where n_{tot} equals the number of users in the datastore.

3) Vertical Search: When users are divided over two datastores, where one datastore ($parent$) is the parent of the other ($child$), the average time needed to find a random user, using a bottom-up serial search, can be calculated as

$$t_{avg} = p_{child} \times t^{(*)}_{child} + p_{parent} \times (t^{(*)}_{child} + t^{(*)}_{parent}) \tag{4}$$

The average time to find a user, using the bottom up serial search, can be divided into 2 subcases: the case where the user is stored at the $child$ datastore (probability p_{child}), and the case where the user is stored at the $parent$ datastore (probability p_{parent}). In the first case, the bottom-up algorithm will only have to search the $child$ datastore. However, in the second case, the algorithm will start searching in the $child$ datastore, and continue the search in the $parent$ datastore.

In case of a parallel search, the average time corresponds to

$$t_{avg} = \max(t^{(*)}_{child}, t^{(*)}_{parent}) \tag{5}$$

Figure 6 illustrates the time needed for finding a corresponding data row in a 2-level hierarchical structure, existing of a tenant (parent) and a subtenant(child), using the *bottom-up* approach. The horizontal axis denotes the amount of data stored at the child node, while the vertical axis denotes the normalised response time, where 1 equals the time needed to find the data in a single datastore. As can be seen from this

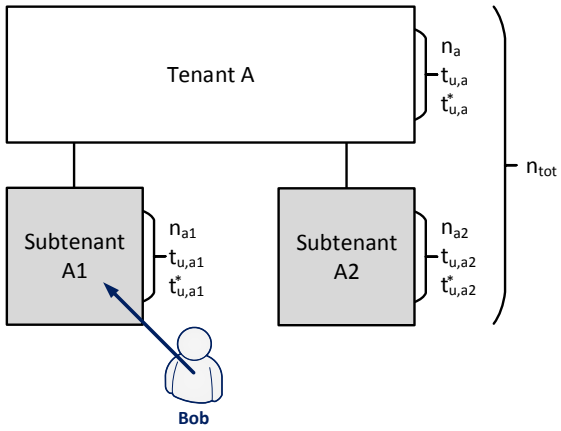


Fig. 7: Extension to the full model. User "Bob" wants to access the data of subtenant A1, and needs a role which can be stored at the subtenant or tenant node.

figure, the distributed serial search on an indexed parameter will have a bad influence on the performance when data is split over different datastores, especially if most of the data is located at the parent node. In all other models, splitting the data over multiple stores results in better performance, with an optimum if the amount of data is equally divided over both stores.

4) Horizontal Search: In case the data is divided over 2 datastores on the same level ($ds1$ and $ds2$), for example in the case of 2 subtenants, we won't have to search both datastores. For example, if we want to find out if user "Bob" has access to subtenant B1, we will only search the roles datastore of subtenant B1, and not the roles datastore of subtenant B2. So, in case there are no parent nodes, and data is divided on the same level, the average time to find a role in datastore $ds1$ can be given as

$$t_{avg} = t^{(*)}_{ds1} \tag{6}$$

5) Impact of other subtenants on the performance: We can easily extend the horizontal and vertical search to a full hierarchical model. Figure 7 shows a simple tree with a single tenant and 2 subtenants. In our example scenario from Section IV, user "Bob" want to access the data of subtenant A1. While the datastore of subtenant A1 is dedicated, the datastore of tenant A is shared between all subtenants. When a subtenant adds extra data to the shared datastore, this will have an impact on the performance. The total number of records in the shared datastore is given as

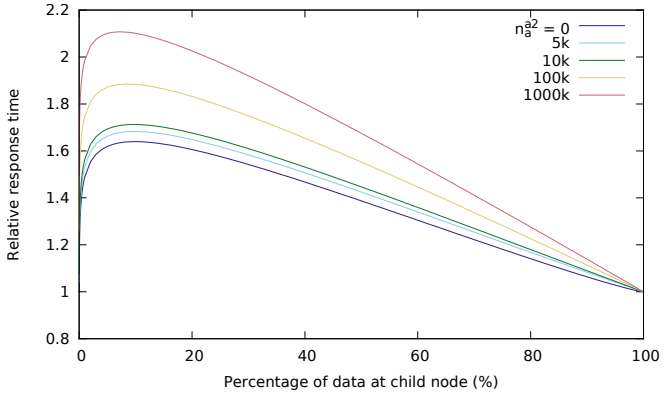
$$n_a = n^a_a + n^{a1}_a + n^{a2}_a \tag{7}$$

where n^a_a denotes the number of records added by tenant A and n^a_a1 and n^a_a2 the number of records added by subtenants A1 and A2. The time needed to find Bob's role equals

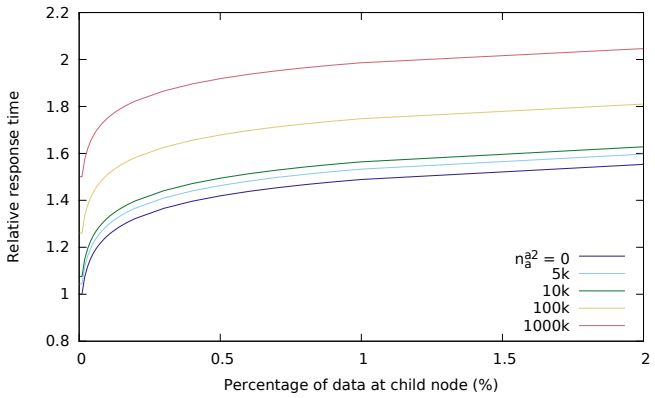
$$t_{avg} = p_{a1} \times t^{(*)}_{a1} + p_a \times (t^{(*)}_{a1} + t^{(*)}_a) \tag{8}$$

with probabilities

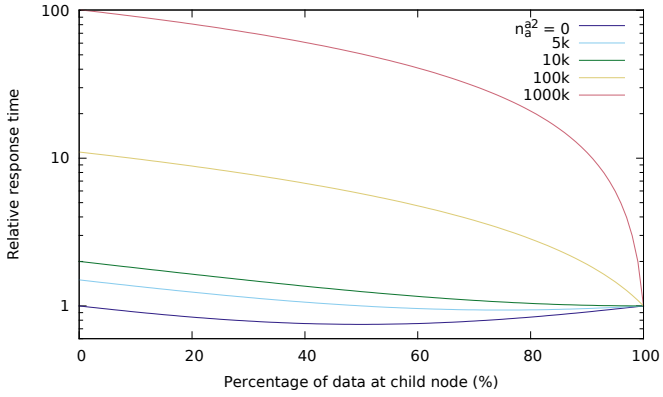
$$p_{a1} = n_{a1}/(n_{a1} + n^{a1}_a) \tag{9}$$



(a) Distributed Serial Indexed Search



(b) Distributed Serial Indexed Search - Detail



(c) Distributed Serial Non-Indexed Search

Fig. 8: Theoretical analysis of the impact of other subtenants on the time needed for finding tenant A1's data in a full model. Tenant A1 has a total of 10k records divided over the 2 datastores. The variable n^{a2}_a denotes the extra rows added to the shared (parent) datastore by the other subtenants.

$$p_a = n_a^{a1}/(n_{a1} + n_a^{a1}) \quad (10)$$

Figure 8 illustrates what happens when the number of records added by the other subtenants (subtenant A2 represents all other subtenants) increases for both the distributed serial indexed and non-indexed search. Note that the different curves in Figure 8a don't start at the same point, as can be seen in Figure 8b. As can be seen from this figure, the influence of the other subtenants on the overall performance of subtenant A1 will decrease as more of the data of subtenant A1 is stored at the leaf node. Hence, when splitting a datastore vertical, it is never a good idea to equally divide the data over the tenant and subtenant.

C. Conclusions

The theoretical analysis shows that in most of the cases, splitting the data over multiple stores won't have a bad influence on the performance of the application. Only when using a distributed serial search in a 2-level hierarchical (vertical) structure, and when searching on an indexed value, the performance will be bad when most of the data is stored in the parent datastore. However, in most structures, the datastores will not only be split vertical, but also horizontal, and most of the data will be located at the leaf nodes, yielding much better performance.

V. Evaluation Results

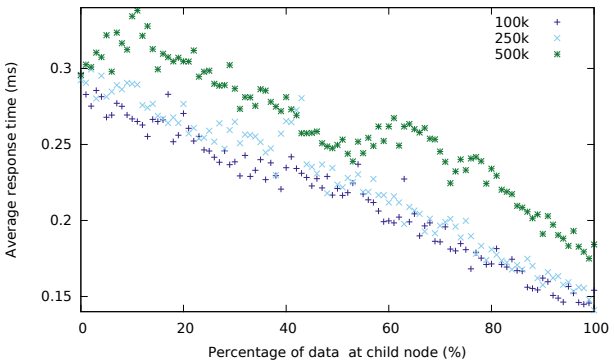
In this section, we will verify our theoretical analysis of the performance by different experiments.

A. Experimental Analysis

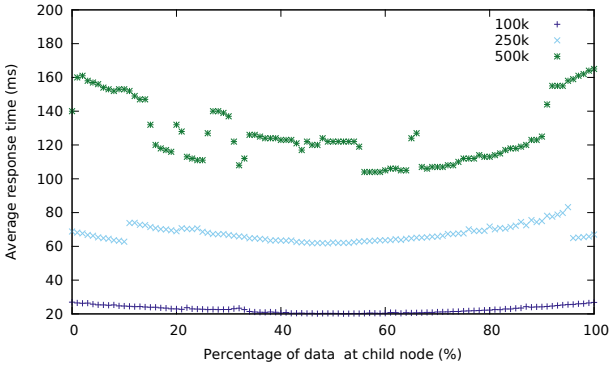
To verify our theoretical analysis, we ran some experiments on 2 different environments. For the first environment, we used a MySQL dbms on Ubuntu 13.04, running on a virtual machine with a single core vCPU and 2GB of memory installed. As a second environment, we configured Microsoft SQL Server on Windows 2008R2, running on a physical machine with a 2.8GHz Intel Core i5 (quad core) and 4GB of memory installed.

During the experiment, we calculated the average time to find a random user in a 2-level hierarchical structure (tenant-subtenant), using the distributed serial bottom-up search. Figure 9 shows the results for both environments, for a total of 100k, 250k and 500k users. During all experiments, the average time was calculated by authenticating 10 percent of the total amount of users. The horizontal axis denotes the percentage of data stored at the child (subtenant) node, while the vertical axis denotes the average response time, expressed in milliseconds.

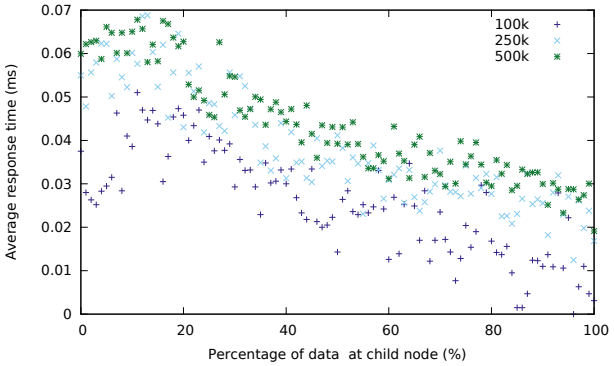
As can be seen from the figure, the experimental results resemble the calculated theoretical times, except for the indexed serial search on the SQL Server environment. This exception is due to the fact that response times are very low when using indexing, making the overhead of searching two databases bigger than the average time. As in the theoretical analysis, the distributed indexed search yields better results as more users are located at the child node. For the distributed non-indexed search, best results are achieved when data is equally



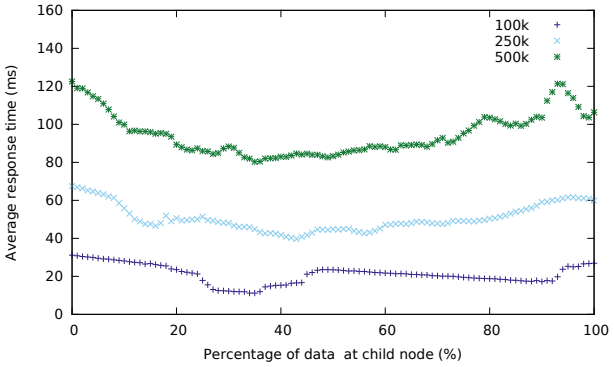
(a) MySQL - Distributed Serial Indexed Search



(b) MySQL - Distributed Serial Non-Indexed Search



(c) SQL Server- Distributed Serial Indexed Search



(d) SQL Server- Distributed Serial Non-Indexed Search

Fig. 9: Experimental analysis of the time needed for finding the tenant data in a 2-level hierarchical structure tenant-subtenant.

divided over both datastores. As the experimental results are in line with the calculated theoretical results, the theoretical analysis can be used to optimize the hierarchical structure for big amounts of data.

B. Conclusions

Experiments confirmed that the results are in line with the theoretical analysis. Therefore, the theoretical analysis can be used to build the decision support part of the the data access layer, for autonomous splitting and merging tenant datastores. In case of a serial search, there is some small overhead due to the switching between datastores, but this overhead can be neglected as the size of the datastore grows. In our evaluation, we only focused on the two subcases (parent-child and 2 nodes at the same level), but this can be used for extension to a full hierarchical structure like the one proposed in Figure 4.

VI. Conclusions and Future Work

As the number of tenants and users in the multi-tenant application grows, users, roles and tenant data can be split over multiple datastores. In this paper, we presented a hierarchical model for the logical representation of the tenant tree and a mapping to the physical storage. Users, roles and tenant data can be divided using the monolithic, the fully distributed or the hybrid model. When data is divided over multiple datastores, we can make use of a serial (bottom-up or top-down) search, or a parallel search when the datastores are located on different machines.

The theoretical and experimental analysis confirmed that the hierarchical model presented in this paper can be used to build autonomous high scalable multi-tenant applications in the cloud. By using the hierarchical model for both the physical representation of tenants and subtenants, and choosing a strategy for the physical storage of the different datastores, it is straightforward to create a mapping between the two models, making the management of the application less complex.

In future work, we will focus on the elasticity of the SaaS application and decision support part of the data access layer, to build an autonomous system for automatic scaling of the application and data in the public cloud.

Acknowledgment

This research is partly funded by the iMinds PUMA[2] project.

References

[1] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao, "A framework for native multi-tenancy application development and management," in *9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007.*, 2007, pp. 551 – 558.

[2] (2013) PUMA: Permissions, User Management and Availability for Multi-tenant SaaS Applications. [Online]. Available: http://www.iminds.be/en/research/overview-projects/p/detail/puma-2

[3] M. Armbrust, R. Fox, Armandoand Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds : A Berkeley view of cloud computing," University of California at Berkley, Tech. Rep., 2009.

[4] C.-P. Bezemer and A. Zaidman, "Multi-tenant SaaS applications: maintenance dream or nightmare?" in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: ACM, 2010, pp. 88–92. [Online]. Available: http://doi.acm.org/10.1145/1862372.1862393

[5] P.-J. Maenhaut, H. Moens, M. Verheye, P. Verhoeve, S. Walraven, W. Joosen, and V. Ongenae, "Migrating Medical Communications Software to a Multi-Tenant Cloud Environment," in *IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, May 2013, pp. 900–903. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6573107

[6] M. Decat, B. Lagaisse, D. Van Landuyt, B. Crispo, and W. Joosen, "Federated authorization for software-as-a-service applications," in *To be published in the Proceedings of DOA-Trusted Cloud'13*, 2013.

[7] M. Decat, B. Lagaisse, and W. Joosen, "Toward efficient and confidentiality-aware federation of access control policies," in *Workshop on Middleware for Next Generation Internet Computing*. ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2405178.2405182

[8] J. M. A. Calero, N. Edwards, J. Kirschnick, L. Wilcock, and M. Wray, "Toward a Multi-Tenancy Authorization System for Cloud Services," *IEEE Security and Privacy*, vol. 8, no. 6, pp. 48–55, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1109/MSP.2010.194

[9] C. D. Weissman and S. Bobrowski, "The design of the force.com multitenant internet application development platform," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 889–896. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559942

[10] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Proceedings of the 29th conference on Information communications*, ser. INFOCOM'10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 534–542. [Online]. Available: http://dl.acm.org/citation.cfm?id=1833515.1833621

[11] S. Walraven, E. Truyen, and W. Joosen, "A middleware layer for flexible and cost-efficient multi-tenant applications," in *ACM/IFIP/USENIX 12th International Middleware Conference*, vol. 7049. Springer Berlin / Heidelberg, Dec. 2011, pp. 370–389. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/313768

[12] E. Yuan and J. Tong, "Attributed based access control (abac) for web services," in *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, 2005, pp. –569.